ISSN 0265-2919

R 80p 5

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

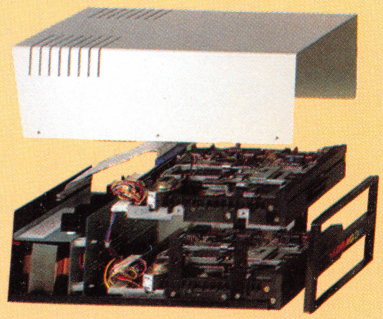
An ORBIS Publication

IR £1 Aus $1.95 NZ $2.25 SA R1.95 Sing $4.50 USA & Can $1.95

CONTENTS

Next Week

- We look at the AIM 65 — a single-board computer designed as a teaching and development aid.
- No matter how small your business there is software available to suit your needs. We start a new series taking a close look at the type of packages that might be of benefit.
- With the introduction of its disk drive, the Dragon becomes a more attractive proposition.



COVER PHOTOGRAPHY BY MARCUS WILSON SMITH

Editor Jonathan Hilton; **Art Director** David Whelan; **Deputy Editor** Roger Ford; **Production Editor** Catherine Cardwell; **Staff Writer** Brian Morris; **Picture Editor** Claudia Zeff; **Designers** Hazel Bennington, Julian Dorr; **Sub Editors** Robert Pickering, Keith Parish; **Art Assistant** Liz Dixon; **Editorial Assistant** Stephen Malone; **Researcher** Helena Siedlecka; **Contributors** Lisa Kelly, Steven Colwill, Richard King, Martin Hayman, Richard Forsyth; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Chris Cooper; **Production Co-ordinator** Ian Paton; **Circulation Director** David Breed; **Marketing Director** Michael Joyce; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 85 Charlotte Street, London W1; **© APSIF Copenhagen 1984; © Orbis Publishing Ltd 1984: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Artisan Press Ltd, Leicester**

HOME COMPUTER ADVANCED COURSE – Price UK 80p IR £1.00 AUS $1.95 NZ $2.25 SA R1.95 SINGAPORE $4.50 USA and CANADA $1.95

How to obtain your copies of HOME COMPUTER ADVANCED COURSE – Copies are obtainable by placing a regular order at your newsagent, or by taking out a subscription. Subscription rates: for six months (26 issues) £23.80; for one year (52 issues) £47.60. Send your order and remittance to Punch Subscription Services, Watling Street, Bletchley, Milton Keynes, Bucks MK2 2BW, being sure to state the number of the first issue required.

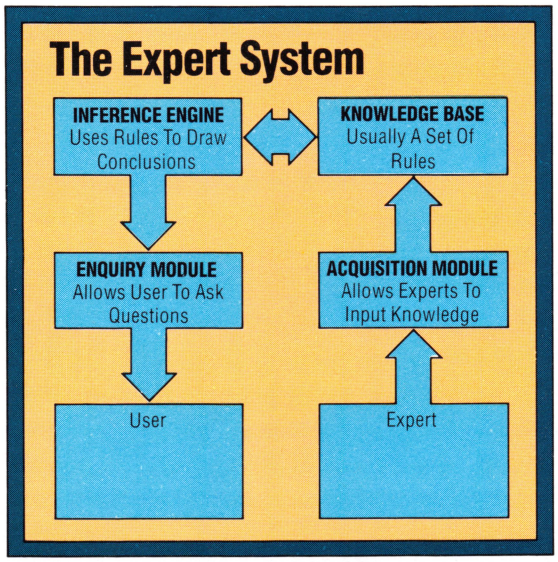
Back Numbers UK and Eire – Back numbers are obtainable from your newsagent or from HOME COMPUTER ADVANCED COURSE. Back numbers, Orbis Publishing Limited, 20/22 Bedfordbury, LONDON WC2N 4BT at cover price. AUSTRALIA: Back numbers are obtainable from HOME COMPUTER ADVANCED COURSE. Back numbers, Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G Melbourne, Vic 3001. SOUTH AFRICA, NEW ZEALAND, EUROPE & MALTA: Back numbers are available at cover price from your newsagent. In case of difficulty write to the address in your country given for binders. South African readers should add sales tax.

How to obtain binders for HOME COMPUTER ADVANCED COURSE – UK and Eire: Please send £3.95 per binder if you do not wish to take advantage of our special offer detailed in Issues 5, 6 and 7. EUROPE: Write with remittance of £5.00 per binder (incl. p&p) payable to Orbis Publishing Limited, 20/22 Bedfordbury, LONDON WC2N 4BT. MALTA: Binders are obtainable through your local newsagent price £3.95. In case of difficulty write to HOME COMPUTER ADVANCED COURSE BINDERS, Miller (Malta) Ltd, M.A. Vassalli Street, Valletta, Malta. AUSTRALIA: For details of how to obtain your binders see inserts in early issues or write to HOME COMPUTER ADVANCED COURSE BINDERS, First Post Pty Ltd, 23 Chandos Street, St. Leonards, NSW 2065. The binders supplied are those illustrated in the magazine. NEW ZEALAND: Binders are available through your local newsagent or from HOME COMPUTER ADVANCED COURSE BINDERS, Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington. SOUTH AFRICA: Binders are available through any branch of Central Newsagency. In case of difficulty write to HOME COMPUTER ADVANCED COURSE BINDERS, Intermag, PO Box 57394, Springfield 2137.

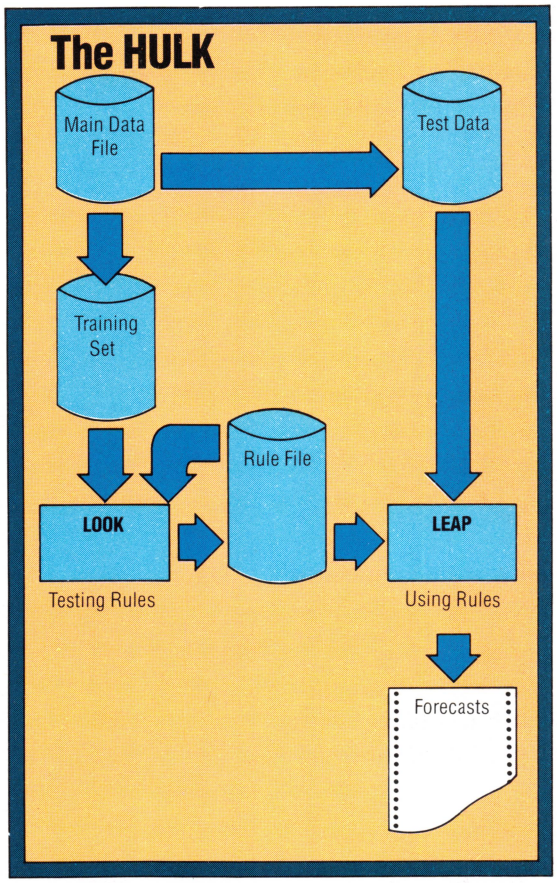
Note – Binders and back numbers are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK market only and may not necessarily be identical to binders produced for sale outside the UK. Binders and issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.

THE AMAZING HULK

The word *engineering* is undergoing a shift in meaning. In the 19th century an engineer was someone like Brunel (1806–59), who transformed iron and steel into ships and bridges. Nowadays the term has been appropriated by a variety of deskbound professions engaged in 'engineering' raw materials such as 'knowledge'.



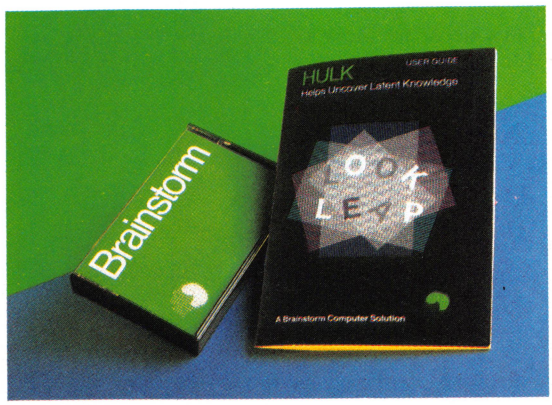
A knowledge engineer is someone who knows how to build an *expert system*. An expert system (often abbreviated to ES) embodies organised knowledge about some area of human expertise. Consider, for example, the case of medical diagnosis: a doctor has a large store of factual knowledge about illnesses and their signs and symptoms — the doctor's expertise lies in the ability to relate a patient's condition to the textbook descriptions of typical conditions. In doing this, the doctor establishes which symptoms are present and weighs their significance against that of the absent symptoms, all in the light of past experience of other patients with these symptoms and/or the suspected illness. The better the doctor is at combining textbook knowledge with actual observations, the better the diagnostic technique will be. The limiting factors are the ability to remember organised data, the ability to relate observed cases to the pattern of existing data, and the ability to apply this knowledge in cases where the data is incomplete or does not quite match previous cases. These first two factors — organisation and classification of data — are what computers are good at; the last factor is what human experts are good at. If a computer system can substitute statistical analysis for the human expert's 'feel' or 'flair', then its superior data-organising powers may enable it to out-perform an expert.



LIZ DIXON

Economy HULK
In a traditional expert system a series of IF. . .THEN rules, held in the knowledge base, are applied by the inference engine in response to users' enquiries entering the system via the enquiry module. The knowledge on which these rules are based is input to the system by experts via the acquisition module. With HULK a set of decision rules are built up from a main data file containing observations input by the user. Two separate programs are necessary: LOOK, which requires a training set of data for trying out rules and a test set of data for testing their usefulness, and LEAP, which uses the rule file and test data to produce a probability forecast. HULK has been described as a poor man's expert system

Such systems are most relevant where human judgement is needed because a complete theory does not exist, as in medical diagnosis. They can also be usefully applied where, even though the requisite knowledge is publicly available and completely organised, it is too complex for most people to apply (for example, tax legislation or the regulations determining entitlement to social



LIZ HEANEY

Practical Project
The HULK was initially conceived of as a practical project by Richard Forsyth for his post-graduate diploma students at the Polytechnic of North London. Although the program was only two weeks in the writing, a further six months were necessary improving and refining it in terms of 'user-friendliness'. A QL version is promised in the near future

security benefits). Because knowledge, frequently expressed as a collection of rules, is crucial to the operation of such systems, they are also known as 'rule-based' or 'knowledge-based' systems.

Expert systems have proved strikingly successful in several fields. Already they can outperform skilled human practitioners in medical diagnosis, prospecting for minerals and in many other fields. Because of these dramatic successes they are emerging from artificial intelligence research laboratories and finding applications in general computing practice, with significant consequences for the way people build high-performance computer systems. The knowledge-based approach to software design replaces the tradition of:

DATA + ALGORITHM = PROGRAM

with a methodology based on:

KNOWLEDGE + INFERENCE = SYSTEM

which is an evolutionary change with very significant consequences.

So an expert system is based on a body of knowledge. There are in fact another three essential components of a fully fledged ES: the 'inference engine', which forms new rules for interpreting data on the basis of existing rules and data; the 'knowledge-acquisition module', through which the system acquires knowledge from its own experience, and from that of human experts; and the 'user interface', which allows non-experts to interrogate and use the system. Let us look at the two core modules — the knowledge base and the inference engine — in greater detail.

A knowledge base contains facts (or assertions) and rules. Facts can change rapidly — for example, during the course of a consultation. Rules are the longer-term information about how to generate new facts or hypotheses from what is presently known. How does this differ from a conventional database? The major difference is that a knowledge base is more creative. Facts in a database are normally passive: they are either there or they are not, to be filed or retrieved by the user as required. A knowledge base, on the other hand, actively tries to fill in missing information in the light of what it already 'knows', and independently of the immediate demands for data.

Rules in the IF...THEN format (known as 'production rules') are a favoured method of expressing 'rule-of-thumb' knowledge. For example:

> IF the home team lost its last home game, AND the away team won its last home game by two goals
> THEN the likelihood of a draw is multiplied by 1.088.

These rules are not embodied in program code: they are data for the inference engine, which is a high-level interpreter.

There are two major high-level strategies for

Weather Forecaster

HULK data files are created by the user as BASIC program files, thus making them easily accessible for inspection and editing. The file we used is typical.

We ran **HULK** using this file, and were asked for a hypothesis about the data; the hypothesis was:

TOMORROW=1

meaning that we are interested in those samples where the last variable has value 1 — in other words, days on which it rained on the following day. We want to use **HULK** to establish rules to enable us to predict from today's weather what tomorrow's will be.

Next we are able to enter rules for testing. Each rule is applied to the data file and its success in predicting rain is measured and reported, both as a single rule, and when used in combination with previous rules. **HULK** advises whether or not to add the rule to the rule set, but the decision is left to the user.

We quickly found three rules for predicting rainfall tomorrow:
1) If it rains more than 2mm today

AND

2) If the sun shines fewer than 3.5 hours today

AND

3) If maximum temperature is less than 6 degrees higher than today's minimum temperature

THEN

you can be 83% confident in predicting rain for tomorrow (given, of course, that this month's weather data is a fair sample of the whole year).

Although this is a BASIC program file, it is not intended to be RUN; it is simply a convenient way of storing data

```
1 WEATHER,30,5
```

Line 1 describes the file: containing its name (WEATHER), the number of data samples (30), and the number of data items (5) per sample. The file contains weather data for one month in London: each day being described in terms of minimum and maximum temperature (in centigrade), daily rainfall (in mm.), hours of sunshine, and a Boolean variable whose value is 1 if it rains the following day, and 0 if it doesn't.

```
100 MINIMUMT
200 MAXIMUMT
300 RAINFALL
400 SUNSHINE
500 TOMORROW
```

Lines 100-500 give variable names to the data in each sample

```
1001 D01,54,110,175,32,1
1002 D02,42,125,041,62,1
1003 D03,76,112,077,11,1
1004 D04,27,105,018,43,0
1005 D05,30,120,000,95,0
1006 D06,44,106,000,55,0
1007 D07,48,094,000,51,1
1008 D08,68,092,055,48,1
1009 D09,64,102,048,41,1
     --------"-------
1028 D28,58,154,000,20,1
1029 D29,67,088,064,42,0
1030 D30,45,096,000,68,1
```

Lines 1001-1030 contain the data, with one sample per program line, and each line starting with a label identifying that sample. Notice that HULK values must be integers, so all data has been multiplied by 10 to preserve precision:

Line 1001, for example, should really read
1001 DO1,5.4,11.0,17.5,3.2,1 and so on for the rest of the file.

TOMORROW = 1

ASKING FOR RULE [1]

RULE IS: **RAINFALL > 20**

CONTINGENCY TABLE	SUCCESS	FAILURE
RULE TRUE	12	2
RULE FALSE	5	11

SUCCESS RATE = **76.3%**

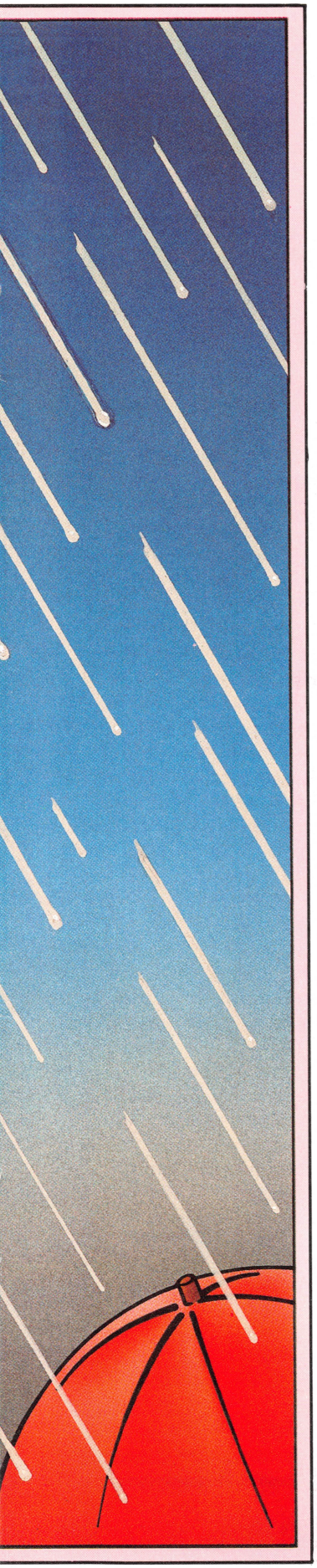
BITS PER SAMPLE

BEFORE	**RAINFALL > 20**	**1.00**
AFTER	**RAINFALL > 20**	**0.85**

YOU ARE ADVISED TO [KEEP] THE RULE OK

RULE HAS BEEN [ADDED] TO THE RULE SET

Bits Per Sample
This is a measure of how many of the data samples are contributing to the success of the rule



MIKE BROWNLOW

drawing inferences: 'forward chaining' and 'backward chaining'. Broadly speaking, forward chaining involves examining data in order to form hypotheses, while backward chaining attempts to find data to prove or disprove an already formed hypothesis. Pure forward chaining leads to unfocused 'What if...' type questioning of the system, whereas pure backward chaining tends to be rather relentless in its goal-directed questioning.

Most successful systems use a mixture of both. Whether an inferencing procedure works primarily backwards or forwards, it will have to deal with uncertain data. Computer specialists have tried to force the world we live in into the rigid confines of the computer, and it has never been a comfortable fit. Now ES research has given us means of dealing precisely with uncertainty — in other words with the real world rather than some idealised abstraction that our data system forces us to use.

Indeed we have too many ways of dealing with uncertainty. There are Fuzzy Logic, Bayesian Logic, Multi-Valued Logic and Certainty Factors to name only four: such logics replace the certainty of 'IF X IS TRUE THEN Y IS TRUE' with the cautious statistical inference, 'IF X IS TRUE 65% OF THE TIME THEN Y IS BETWEEN 50% AND 70% LIKELY'. All sorts of schemes have been tried, and the odd thing is that they all seem to work. A possible explanation of this state of affairs is that the organisation of knowledge matters more than the numeric values attached to it. Most knowledge bases incorporate redundancy to allow the expert system to reach correct conclusions by several different routes. The numbers measuring degree of belief are important only as a way of choosing between one value set and another.

Software packages designed to facilitate the knowledge-based approach to systems design are now coming onto the market. So far the only one within the budget of the home computer enthusiast is the HULK (Helps Uncover Latent Knowledge), on sale from Brainstorm Computer Solutions at around £25. It runs only on the BBC Model B and Torch computers, although a QL version is promised.

HULK enables the user to build up and test a set of decision rules, which can later be used for prediction or classification. You will find it useful if you have a set of examples or cases each measured on several variables and wish to discover patterns and regularities for predictive purposes.

For instance, suppose a farmer kept detailed measurements on a home computer of the height, leaf colour and so on of several hundred sugar beet plants, and recorded those that became diseased before harvest time and those that remained healthy. The system could be used to help develop rules relating the characteristics of the plant to its state of health. Later those rules could identify plants at risk and thereby improve prospects for the following year's crop. This farmer might never know what those rules were, but the system could apply them to the plant data as it was supplied.

Alternatively, you might hold information on a season's football results and want to develop a way of categorising games as likely wins, draws or losses on the basis of various indicators known before kick-off. The necessary data for making decisions, such as the home team's recent performance, previous results of this fixture, relative positions of the teams in the league table, is all readily available. But it's very time consuming to organise and difficult to correlate without the aid of an ES. There are plenty of applications for the HULK — all you need is a set of data.

The HULK package consists of two main programs: LOOK (Logical Organiser Of Knowledge) and LEAP (Likelihood Estimator And Predictor). These allow the user to develop a set of rules from a data file of observations held on disk (for example, results of past matches and any other decision factors), and then to apply that rule set to another data file of incomplete observations (for example, the decision factors for next Saturday's matches) in order to make predictions about them. The rule set is the knowledge base: it expresses the system's knowledge about the data. In HULK, it consists of probabilistic decision rules that can be used for classification or forecasting.

The knowledge base grows through the interaction of the user and the computer: rules are proposed by the user and tested by the computer; the user discards those that do not improve the overall performance of the system.

To use LOOK you need a data set, or preferably two (a large data set can be split into two parts for this purpose). One is called the 'training set' for trying out rules, and the other is called the 'test set' for confirming their usefulness on unseen data

Once the data is on file, LOOK is used to test your hunches by proposing new rules, one at a time. It runs each rule over the training set and tells you how much, if at all, it improves the prediction score of the rules already present. Then it recommends whether to keep or discard the new rule. The final decision is left to the user.

LOOK is not a true learning program, but a kind of filter that allows only useful guesses or conjectures through. In a sense, it is a learning program without a learning algorithm; the user does the learning, while the machine does the clerical work of comparing, evaluating, and assessing the data set.

Having created the rules with LOOK, LEAP can be used to apply them to unknown samples. The rules are combined to give a single probability estimate for every sample in the test data set. LEAP's output includes a list of samples ranked according to their likely outcome. What this outcome might be (SCORE DRAW, or HIGH YIELD FROM THIS VINE) depends on the nature of the sample data, and what the user was trying to predict from it. HULK will supply the acquisition module and knowledge base facility, leaving the user in control of the inference engine.

CORPORATION CONTROLLERS

The BBC computers incorporate a flexible method of expansion to include disk drives. As Acorn has supplied details of both the DOS — which it calls a Disk Filing System (DFS) — and the disk controller to many disk drive suppliers, a number of different disk drives are available that are compatible with the BBC Micro.

At present the disk drives compatible with the BBC Micro are virtually all standard 5¼in mini-floppies, ranging in price from around £250 for the smallest-capacity single drive to £600 for the largest-capacity double drives, but 8in drives are becoming available. Unfortunately, DFS is not fitted as standard on the BBC computers and it must be purchased separately from Acorn for about £80. It is supplied in the form of a ROM chip that is fitted in a socket on the computer's PCB. Other manufacturers offer alternative DFS chips for a little less than Acorn, but the cheapest disk system requires an outlay of about £300.

Because DFS is contained in ROM, most disk commands do not require internal memory to carry out instructions. DFS is extended by commands and routines supplied as 'utilities' on a disk that comes with the drive. These are particular to the disk drive being used and relate to formatting and verification.

Disk drives are attached via ribbon cable to a 34-way connector (marked 'Disk Drive') on the underside of the computer. This cable carries all disk data to and from both double and single drives. Control of disk drives is exercised via an 8271 disk controller chip that converts eight-bit parallel data from the computer to serial form for output to the selected drive, and vice versa. Four standards of format are supported: 40-track or 80-track single-sided and 40- or 80-track double-sided, all at single density. Each track is divided into 10 sectors, containing 256 bytes each and giving a total capacity of approximately 100 Kbytes per side on a 40-track format and 200 Kbytes per side for 80-track.

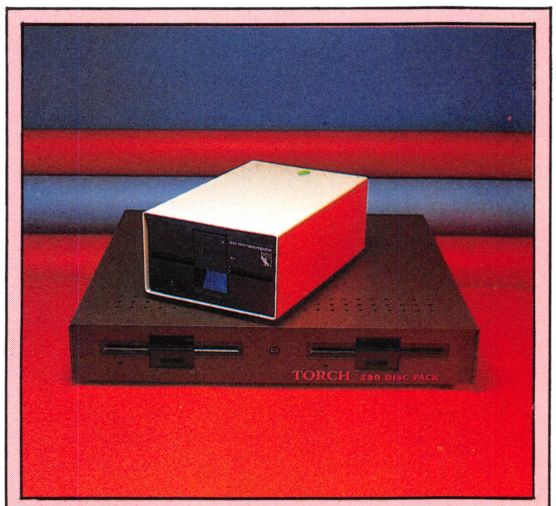
Drives are identified by number, where a single drive is 0 and a second drive is 1. In the case of drives that use both sides of a disk, the two sides are treated as separate drives, numbered 0 and 2. A second drive numbers its two sides 1 and 3.

Each disk (or side) catalogue takes up two Kbytes and is contained in the first two sectors of the first track. The catalogue holds information about the disk name and identifiers and can be divided into as many as 27 separate selectable directories containing lists of the files that they account for so that files can be stored by category. No Block Availability Map (BAM) is held; instead there is a *COMPACT command that locates any gaps left by deleted files and rearranges the storage of files into consecutive order, leaving all free space after the end of the last file.

Program and data files can be saved to disk in the same manner as to tape. In fact, with DFS all cassette-handling commands are automatically routed to disk from power-up. To use a cassette, with DFS fitted, enter RETURN — *TAPE. To reassign to disk enter the command: RETURN — *DISK. In addition to the standard file-handling commands DFS provides many advanced data-handling commands and routines, including a sophisticated random access filing system, HELP commands and error messages.

Like the excellent BBC BASIC, DFS is a powerful device for manipulating data. Many routines usually considered beyond the scope of home computer Disk Operation Systems are included and they are, moreover, easy to use. Efficient disk management is encouraged by the structure of the system and data transfer is fast. Figures vary from manufacturer to manufacturer but on average it takes about five seconds to load a 20 Kbyte program file. The drawback of the system, apart from its relatively high cost, is that DFS can store only 31 file names per disk side. Bearing in mind the possible capacity of 200 Kbytes per side and the highly flexible directory system, this is a real limitation.

Lord Of The Strings
The Hobbit is a dedicated floppy tape system designed for the BBC Micro. Being completely under software control all wind, rewind, play and record functions are performed automatically



BBC Disk Drives
To get the most from the BBC Micro a disk drive is essential. The two illustrated here are among the most popular for the Model B — the Acorn 100 and the Torch Z80. Before either can be used they must first be interfaced with the computer via Disk Operating System ROMs, which have to be installed inside the machine

IAN McKINNELL

BBC Disk Commands

In all cases where it is necessary to state the file to which a command relates the full specification is:

COMMAND:DV.DR.FILENAME

where: COMMAND is the required file-handling command; DV is the drive number (0-3); DR is the directory identifier (A-X or $); and FILENAME is up to seven characters that identify the file (these must not include '#', '*', '.' or ':'). If no drive number and directory identifier are specified, the default drive is 0 and the default directory is $. The default identifiers can be changed using the *DRIVE, *DIR & *LIB commands. DFS also allows the use of these 'wildcard' characters: '#' and '*'. In some commands these can be used to specify all drives, all directories or all file names starting with the same letter. In the following explanations the above file specification is abbreviated to <FSP>.

In addition to the standard cassette commands available for disk from power up and the random access commands, DFS provides the following disk-handling commands:

***FORM40, *FORM80 And *VERIFY**
These commands are stored as utilities on a disk supplied with the disk drive or drives; they format a 40- or 80-track disk, and report on the success of the operation.

ACCESS
*ACCESS <FSP> L protects the file from deletion or overwriting. *ACCESS <FSP> removes this protection.

***BACKUP, *DESTROY And *ENABLE**
*BACKUP SourceDV DestinationDV copies the entire contents of the disk in the source drive to the disk in the destination drive, thus overwriting the destination disk. *DESTROY <FSP> deletes a file. If wildcards are included in <FSP>, then several files may be deleted by the one command. Because BACKUP and DESTROY are so drastic in their effects, you must issue an *ENABLE command before the DFS will obey them.

***BUILD <FSP>**
This creates an ASCII file with the file specification from all subsequent keyboard entries until terminated by the ESCAPE key.

***CAT DV**
This displays the catalogue of the specified drive.

***COMPACT DV**
This moves all spare space on a disk in the specified drive to the end of the last file in a continuous block.

***COPY**
Copies a specified file or files (using a wildcard file name) from one disk to another.

***DELETE <FSP>**
This deletes the single specified file from the catalogue of a disk. The file can then be overwritten.

***DIR DR**
This sets the current default directory to that specified. All subsequent files stored using *SAVE or SAVE will be assigned to the set directory.

***DRIVE DV**
This sets the current default drive.

***DUMP <FSP>**
This displays a listing in hexadecimal of a specified file.

***EXEC <FSP>**
This reads all data from a file as if it were input from the keyboard. It is useful for executing an often-used sequence of commands. Files read by *EXEC are created using *BUILD.

***HELP**
With reference to disk drive operation *HELP DFS displays a partial list of standard DFS commands and their construction; and *HELP UTILS displays a list of the remaining standard DFS commands.

***INFO <FSP>**
This displays extra information concerning the specified file or files (using wildcards) not displayed by *CAT such as: memory location; execution address; length in bytes; and sector location.

***LIB:DV.DR**
This sets the specified directory as the 'library'. It allows the use of a short form of command — * FILENAME — that searches the current library directory for the named machine code program, loads it into memory and executes it immediately as if the full *RUN command had been used.

***LIST <FSP>**
This displays the specified ASCII file, including line numbers.

***LOAD <FSP>**
This reads the specified file into memory at the locations from which it was originally taken.

***OPT 1**
This enables a message system where the information given by *INFO is displayed every time a file is accessed. This facility is enabled by *OPT 1 1. To disable this feature use *OPT 1 0.

***OPT 4**
This changes the auto-start option on power up or (SHIFT) BREAK for the current selected drive where: *OPT 4 0 disables auto-start; *OPT 4 1 LOADs the file !BOOT; *OPT 4 2 RUNs !BOOT; and *OPT 4 4 EXECutes !BOOT.

***RENAME <old FSP> <new FSP>**
This command changes a file name and moves it to a different directory. It cannot move files from drive to drive.

***RUN <FSP>**
This reads a machine code file into memory and executes it immediately. It is used on files not contained in the current library.

***SAVE**
This copies a specified block of computer memory and writes it to disk in the current drive and directory. Constructed as:

*SAVE "NAME" SSSS FFFF EEEE RRRR

or

*SAVE "NAME" SSSS+LLLL EEEE RRRR

where SSSS is the start address of memory block; FFFF the finish address of memory block; EEEE the execution address of stored program; RRRR the address where the program will be read to; and LLLL the length of file in bytes (option to FFFF). All numbers are given in hexadecimal. RRRR and EEEE may be left out, in which case the reload and execute addresses default to SSSS.

***SPOOL <FSP>**
This opens the specified file to receive all information displayed as a text file. It allows a BASIC program to be stored as an ASCII file instead of being tokenised.

***TITLE "DISK NAME"**
Changes the name of the disk in the current drive to the specified name.

***TYPE <FSP>**
This displays an ASCII file excluding line numbers.

***WIPE <FSP>**
Identical to *DESTROY except that *ENABLE is not necessary.

BENCH TEST

The final stage of any assembly operation is its testing – and that does not mean switching it on to see if it works. One cross-connection could mean the destruction of one of the more delicate devices. A simple continuity tester will cost you under £1 to make yourself, and a sophisticated multi-mode tester will be less than £10.

The first stage in the testing of any assembly of components into a working circuit is concerned with the efficiency of the soldered joints. A joint made at too low a temperature may look acceptable externally, but inside it may be making no connection at all. A gentle pull may reveal this – if the joint was properly made it won't come apart in your hands, and if that does happen, better it should fail at this stage.

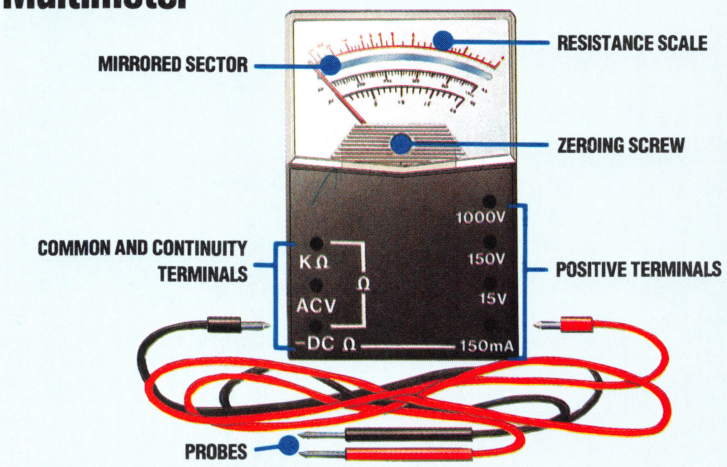
Now we are ready to apply a tester of some sort. The simplest sort of contact tester is available from hardware and car accessory shops. Sold as an 'ignition probe' or some similar name, it is used for determining when the contact points in the distributor are open and closed.

For our purposes, the crocodile clip usually found on one end of the probe is not terribly useful, so fix a spare soldering iron tip into it with insulating tape to use as a probe. Of course, it's an easy matter to make up a tool to do the same job using a couple of 1.5 volt dry cells, a three volt bulb and a length of wire.

A better alternative, however, is a small multimeter. In its role as an ohmmeter, it can test not only continuity but also resistance. The unit of resistance is called an 'ohm', named after the 19th-century German physicist, Georg Ohm (1787-1854), who discovered the phenomenon. Resistance is a function of the cross-sectional area of the wire carrying a current, but it can also be introduced artificially by the use of components known as *resistors*. For our task, a properly made joint will offer negligibly small resistance to the passage of the small current used by the meter or continuity tester, and so the light will come on strongly, or the needle on the dial deflect fully. Any lesser reaction is an indication of a poor joint that should be remade.

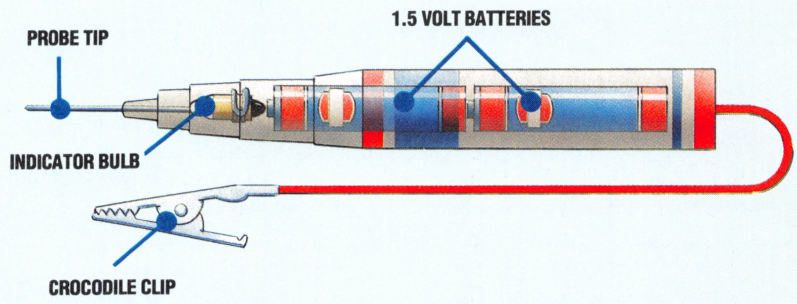
In addition to measuring conductivity, the multimeter has two other functions: the measurement of current in amperes, and of electrical potential in volts. These two units are closely related — the potential difference between two points on a circuit carrying one ampere of current and dissipating one watt of power is one volt.

Multimeter

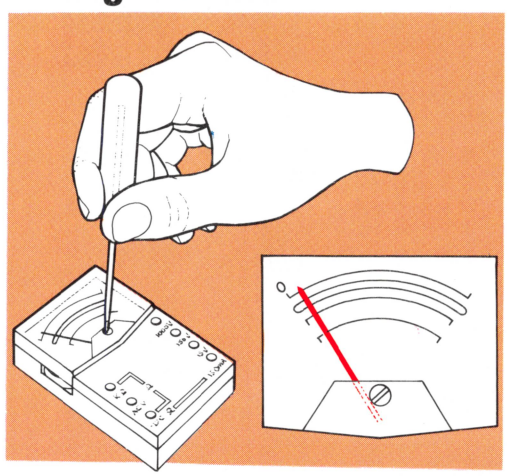


Multimeters — which typically test continuity and resistance, measured in ohms; the magnitude of the current flowing, measured in amperes (amps); and the amount of potential (sometimes described as 'pressure'), measured in volts — vary in price from less than £10 to many hundreds of pounds. However, there are only two methods of representing their results — analogue or digital. In general, moving-coil instruments that display their data by moving (deflecting) a needle across a scale in a manner analogous to the increase or decrease of the value being measured, are considerably cheaper than digital versions

Circuit Tester

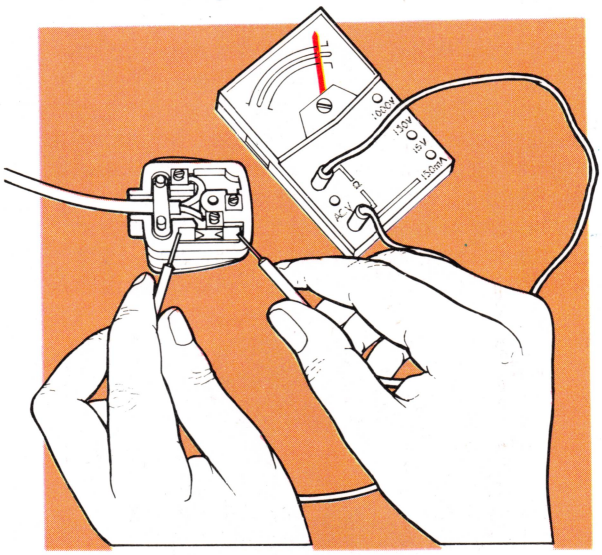


Zeroing The Meter



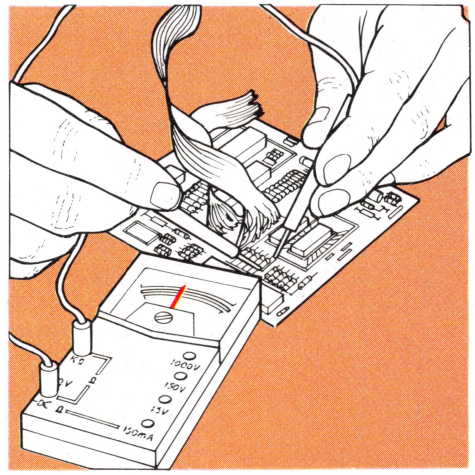
Zeroing The Meter
Analogue meters, which require the indicator needle to move physically across a dial, must have some provision for adjustment. This is normally in the form of a screw, mounted at the fulcrum of the needle. A mirrored sector behind the needle allows the observer to be sure of an accurate reading. In addition, the resistance measuring circuit must also be adjustable to take account of other variations and inaccuracies

Continuity

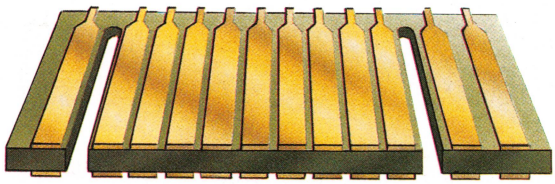


Continuity And Resistance
One of the most common examples of induced deliberate continuity failure is in the fuse found in all our three-pin 13 amp plugs. The fuse is designed to burn through and break the circuit if it gets overloaded, and in the absence of a continuity tester of some description the usual remedy for an appliance's failure to operate is to swop the old fuse for a new one. But what if it wasn't the fuse? The continuity tester will soon tell us. In addition, we can use a multimeter to measure the value of a resistor

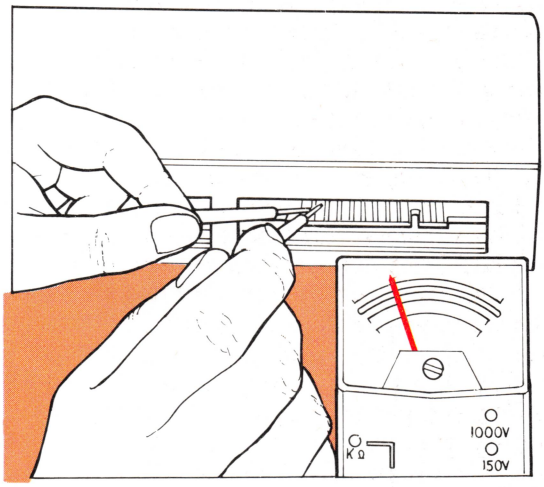
Resistance



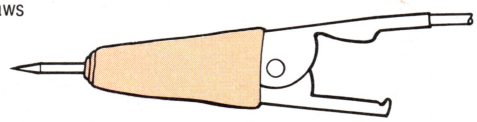
Leading Edge
Edge connectors of the sort shown here are the physical medium by which the computer is connected to its peripherals. Some machines, such as the Spectrum and ZX81, have only this one port. Others, like the BBC Microcomputer, have many. Edge connectors are only one type of I/O port, but are popular because they are an integral part of the PCB



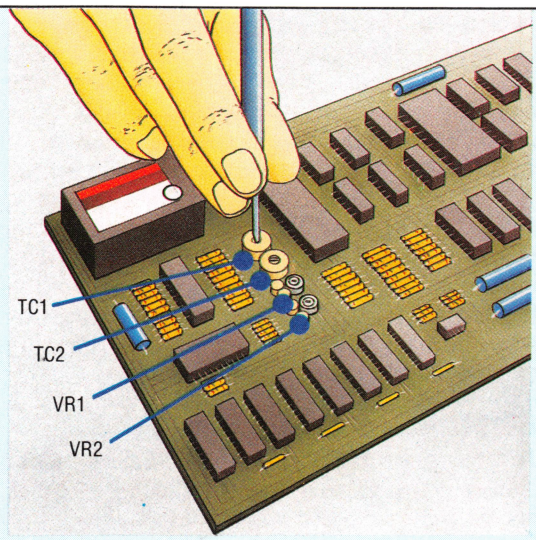
Voltage
A good example of the use of the multimeter as a voltage tester is to locate the power source of your computer's edge connector. Refer to your manual, and locate the 0 volts and +5 or +9 volts pins. Touch the negative probe of the meter to 0 volts and the positive to +5 or +9 volts. The resultant reading will tell you how accurate is the voltage regulation and control circuitry inside your computer and its power supply



Crocodile Clip
The rather imprecise crocodile clip can be converted to a finer probe by taping a spare soldering iron bit between its jaws



WARNING
Your home computer's guarantee — if still in force — may be rendered null and void if anyone other than the manufacturer or his appointed agent opens the case



Tuning Your Spectrum

As we noted on page 50, owners of early Spectrums can improve the quality of a deteriorating TV image. The two variable resistors shown (VR1 and VR2) control the red-green and blue-yellow balance respectively. TC1 (a trimming capacitor) and its companion TC2 control character clarity and colour strength.

The Spectrum case is held together by the five visible cross-head screws on the underside — note the warning about guarantee nullification — and exposing the PCB is simply a matter of removing them.

Version 3 Spectrum owners cannot make this adjustment — or any other. A Version 3 Spectrum can be identified by the heatsink, visible as a thick aluminium plate to the left of the edge connector

ARCHITECTURE

In everyday use, the word *architecture* refers not only to the aesthetics and physical form of a building, but its internal function as well, though it may be the former that immediately springs to mind. In computer terminology, the term refers solely to the *internal* structure and functioning of the machine. The architecture of a microprocessor is primarily defined by the nature of its internal registers and the way in which they can interact with each other, though it can also refer to the way in which that chip is constructed from layers of semiconductor materials. Furthermore, when applied to a microcomputer as a whole, the architecture is a description of the chips contained in that system, and the way in which the available memory space has been allocated to different processing functions. From the user's point of view, those machines with a common architecture are also capable of extensive compatibility in terms of both their hardware and software.

ARRAY

Many computer terms have quite sensible origins in plain English, and *array* is no exception. We might talk about an array of lights on a dashboard, or something being 'in disarray'. An array is a collection of terms with something in common that are arranged in some orderly fashion, usually in the form of a grid with rows and columns. In computing we are mostly dealing with data, and an array of data is the computer equivalent of a table. If it is a two–dimensional table then that simply means that it takes two variables or indexes to specify the piece of data that you want. But computers are equally happy with a one–dimensional array (sometimes called a 'dimension' because in BASIC it is defined by a DIMension statement) which is just like an ordinary list of values.

Non-mathematicians have difficulty grasping the idea of an array of items extended to more than three dimensions, but while some home computers stop at two, others would allow collections of data that needed as many as 13 specifications to uniquely define a single entry.

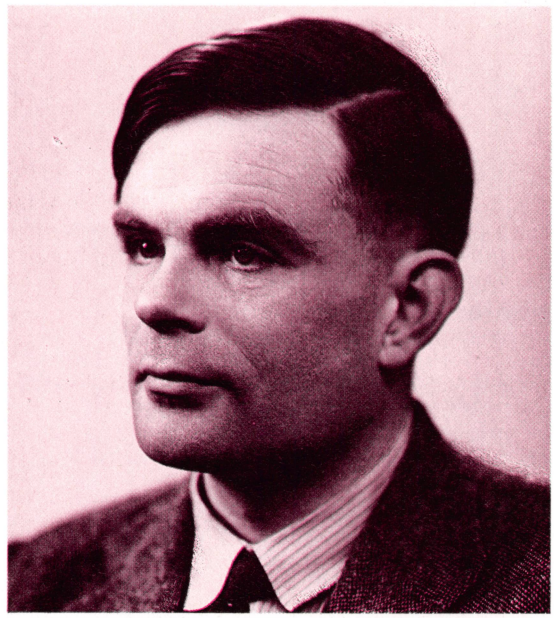
The word *array* is sometimes found in computer hardware specifications, too. Some years ago a great deal of research was being done into the Distributed Array Processor, a device that featured a collection of conventional processors connected into one large grid, so that far more data could be processed simultaneously.

ARTIFICIAL INTELLIGENCE

To many people the whole concept of an intelligent machine is a contradiction in terms. Surely, they feel, computers cannot think, they merely process information that has been fed to them by a human being, and according to a processing procedure (the program) determined by a human being.

However, programs with some learning capacity have been around for some time now, including chess programs that after playing a vast number of games will always beat the programmer. Other programs are self-modifying, to the point that if you were to list the code after several runs, it might have become unrecognisable. Whether such tricks constitute intelligence is hotly debated but there is certainly a large academic community devoted to research into the field, now generally known as AI.

The accepted theoretical test for an intelligent machine was proposed by the British mathematician Alan Turing in the 1950s. A man is



ALAN TURING (1912-1954)

placed in a room that features two teletype machines — one communicating with a second man, the other with the machine undergoing the test. If, having put whatever questions he liked, of whatever kind, the first man was unable consistently to state which teletype communicated with the second man, and which with the computer, then there would be no grounds for stating that the machine was not intelligent.

Most of the fruits of research into AI have been very limited in application, but have included speech and visual pattern recognition, as well as *expert systems*. The latter attempt to model a particular field of expert knowledge, such as diagnosing respiratory diseases or predicting the location of oilfields.

Within AI there are two distinct approaches. The 'top-down' approach attempts to imitate what we would call intelligent behaviour (such as the handling of language) on standard computers. The 'bottom-up' camp, however, argue that we will never be able to emulate the human brain's function using this type of hardware. The latter are trying to develop the electronic equivalent of the neuron (from which our brains are constructed). These devices, when connected together to form a large network, show a tendency to learn from the environment in which they are situated, in a fashion not dissimilar to the way in which we learn.

PERSONAL COMPUTING AT A PRICE

It could be argued that the microcomputer revolution didn't really begin until IBM joined in. IBM is without doubt the largest manufacturer in the world of office equipment and computers, but it did not enter the microcomputer market until 1981. Such was its impact, however, that things have never been the same since.

IBM'S reputation in the mainframe and minicomputer markets is certainly not that of an innovator in hardware. Its software and documentation, though often enormously detailed and accurate, could be improved in terms of presentation and ease of use. However, the company is known for the robustness of its equipment, and the IBM PC is certainly sturdy. Like most IBM hardware, it also costs considerably more than its competition.

This doesn't seem to discourage sales, though, and despite prices that make other home computers seem cheap, the machine soon became one of the most popular. It has been paid the ultimate compliment of being mimicked and cloned at least as much as the Apple, and certainly more quickly.

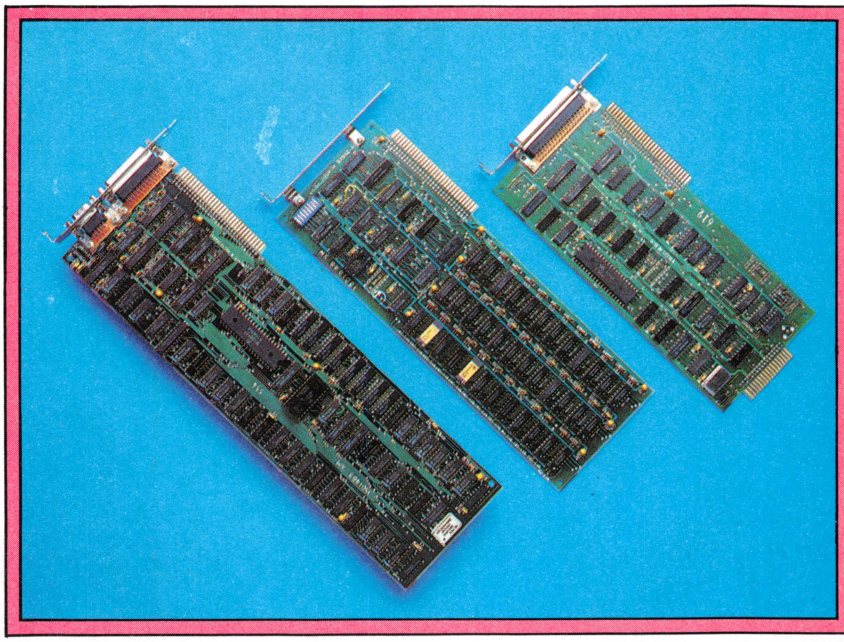
IBM explains that the PC's high price is a reflection of the level of support that the company provides. Support is indeed available — if you are prepared to pay 11.2 per cent of the cost of the item per year for a service contract. The cost of most independent service contracts is at least two per cent higher, and few companies can offer exchange units at a moment's notice, so perhaps there is some additional merit in buying machines built by a very large company such as IBM.

The IBM PC's specification isn't outstanding. It has an 8088 processor, which is described as a 16-bit CPU, but has the data and address lines multiplexed to save pins on the chip, and this means that it isn't fast. In fact, it generally performs only about 25 per cent faster than the average eight-bit machine.

As supplied, the basic model needs some expansion before it fulfils its potential, since it doesn't have much memory (not enough to run complex programs) and it has almost no input and

CHRIS STEVENS

Ergonomical Design
The physical design and layout of the IBM PC reflects the company's huge experience in the field of computers and office products — unobtrusive and ergonomically sound. The three main units — keyboard, processor and monitor — are separated for ease of positioning. The machine comes with a monochrome unit as standard, but a full colour monitor is available



Expansion Boards
In its most basic form, the IBM PC hardly challenges much longer established machines such as the Apple II, but when expanded by the addition of (left to right) a colour graphics board, a memory expansion board and a sophisticated input/output controller, the machine starts to look much more like a serious personal computer for business use

output facilities. As a result most purchasers find that they have to add at least a multi-function card, which costs as much as many home computers.

The graphics are impressive but are not supplied with the machine; the user needs a graphics card. This is available in two versions — monochrome and colour — and consists of a large block of memory (to hold the screen image) and electronic components that produce the video signal. The cards are controlled by the main CPU, although a new type of display card is emerging that has a specialised video display processor. This frees the main processor from the task of updating the screens and so speeds up the process. It is, unfortunately, very expensive.

The PC has expansion slots, but there are only five and so careful thought has had to be given as to how they are to be used. The result is that virtually no 'cheap and cheerful' cards offering limited functions at a low price are available — almost all of them are large, complex and capable of performing many different jobs, often simultaneously, and they are expensive. Since the IBM PC isn't much use without such enhancements, their cost should be taken into account when considering purchase of the machine.

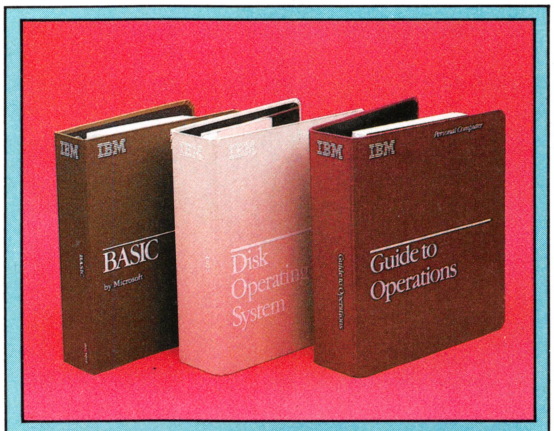
There are, of course, alternative models of the PC — such as the hard-disk XT version — which have many of these facilities provided as standard, but the prices are much higher. In fact, they are comparable with that of Apple's Lisa, a much more advanced machine.

Proving that even IBM isn't immune to fashion, the PC has been revamped as a 'portable', but at a minimum weight of 14kgs (30lbs) this is stretching the term. The remodelling involves replacing the two standard-height disk drives with the double-sided drive, which then leaves one of the apertures available for a 9in amber monitor. A lighter and smaller version of the keyboard clips over the face of the machine, and it is housed in a new case.

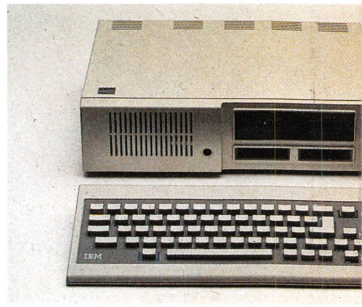
Disk Drives
The machine comes fitted with only one single-sided, single density disk drive, but this can be progressively upgraded to double-sided double density

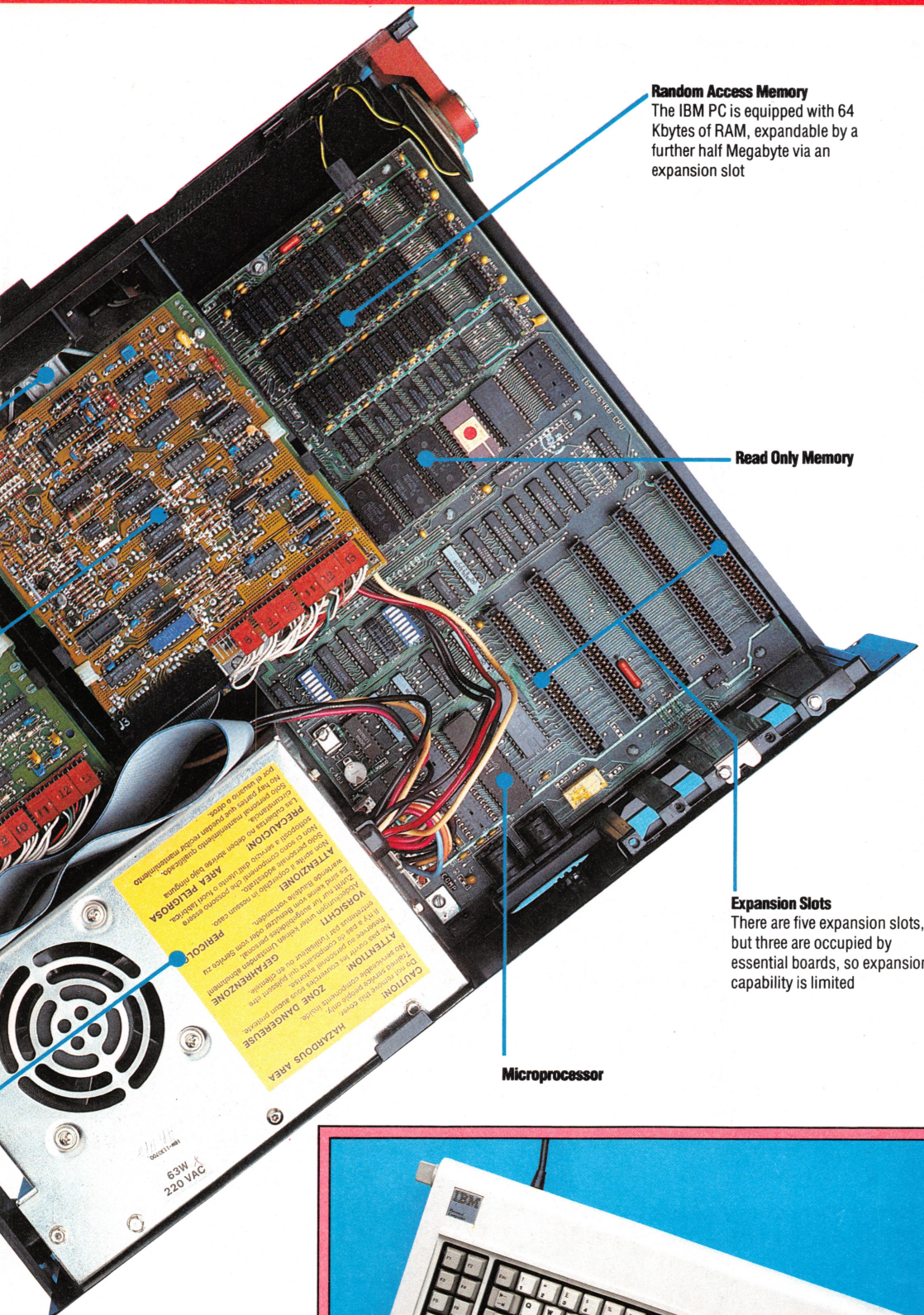
Disk Controller Boards

Power Supply
The IBM PC uses Intel's 8088 microprocessor, which has 16-bit addressing but only 8-bit data transfer



IBM PC Software
One of the chief reasons for buying an IBM PC — and the major reason why so many computer manufacturers around the world have copied it almost exactly — is the choice of commercial software available. It runs under the control of PC-DOS (disk operating system), developed by Microsoft on the basis of CP/M, though a number of alternatives are available — CP/M-86 and the UCSD p-system, for example — and between them these operating systems support a wide variety of languages, such as COBOL, FORTRAN, PASCAL and BASIC. The range of commercial software is as broad as that for any microcomputer system available, and includes a variety of word processors, spreadsheets, databases and business packages. In addition, because the IBM PC in the United States is regarded as much more of a home computer than it is in the rest of the world, there are a large number of games from US software houses





Random Access Memory
The IBM PC is equipped with 64 Kbytes of RAM, expandable by a further half Megabyte via an expansion slot

Read Only Memory

Expansion Slots
There are five expansion slots, but three are occupied by essential boards, so expansion capability is limited

Microprocessor

IBM PC

PRICE
£2,033

DIMENSIONS
140×500×400mm

CPU
Intel 8088

MEMORY CAPACITY AND SPEED
64K RAM, expandable to 576K.
40K ROM.
4.7MHz

SCREEN CHARACTERISTICS
25 lines of 80 characters

INTERFACES AND PORTS
Centronics parallel and five slots

AVAILABLE LANGUAGES
BASIC, plus a selection available under PC-DOS, including COBOL, FORTRAN etc.

KEYBOARD
79 typewriter-style keys

DOCUMENTATION
Up to the normal standard expected from IBM and its chosen software suppliers. Independent software documentation is varied in its quality

STRENGTHS
The basic IBM PC can be used quite successfully, and then expanded and upgraded into one of the most powerful microcomputers available. And it comes from the world's largest computer maker

WEAKNESS
It is certainly very expensive when compared to the many PC-lookalikes that followed its release. Because it is a complex piece of equipment, service and software are also more expensive than for simpler machines

The Peanut
IBM's PC Junior, codenamed 'Peanut' during development, is a severely downgraded version of the larger machine. Perhaps the most exciting innovation was its use of an infra-red link between keyboard and processor, instead of the usual cable. Staking a claim to be a home computer proper, the PC Jnr is equipped with two cartridge slots

Keyboard
As might be expected from one of the world's largest manufacturers of typewriters, computer terminals and other keyboard-driven devices, the keyboard of the IBM PC is virtually fault free. It is detached from the processor unit, so as to be positioned to suit the user, is extremely flat in profile and adjustable for rake, and uses sculptured keys arranged in five rows of keys that form a sector of the circumference of a large drum

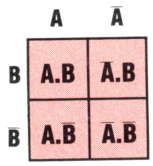
CHRIS STEVENS

MAP READING

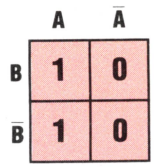
Karnaugh maps are extremely useful devices in the simplification of logic circuits. Whilst not entirely replacing the algebraic simplifications that we have investigated previously (see page 47), they do dispense with much of the effort involved in the factorisation of complicated Boolean algebra expressions.

Karnaugh maps (also known as *k-maps*) are really extensions of the Venn diagrams that we encountered earlier in the course (see page 46), which allow us to represent logical expressions pictorially. A k-map takes on slightly different forms depending on the number of different letters (or variables) there are in the expression to be simplified, but is most useful for expressions containing two, three or four variables.

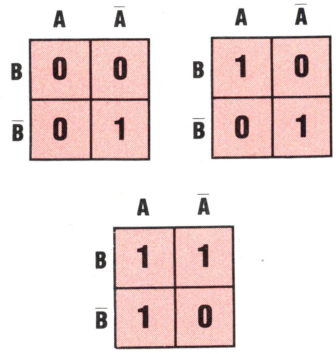
Two Variables: Each square of a two variable k-map (there are $2^2 = 4$ squares) represents an ANDing function, as shown in this diagram:



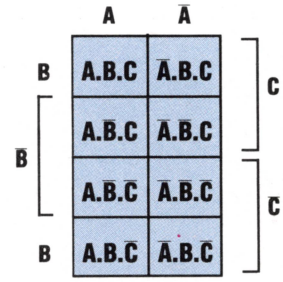
To represent the expression $AB + A\overline{B}$ as a k-map, we place ones in the relevant squares:



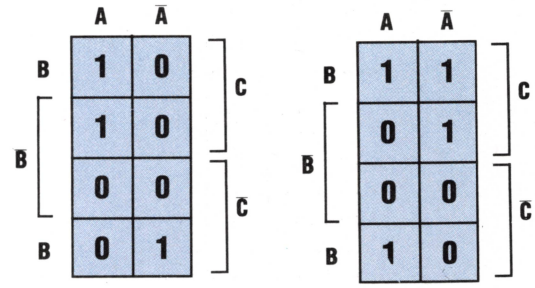
Here are three further examples, representing the expressions $\overline{A}\,\overline{B}$, $AB + \overline{A}\,\overline{B}$, and $AB + \overline{A}B + A\overline{B}$ respectively:



Three Variables: In this case, the number of squares increases by a factor of two ($2^3 = 8$ squares). The basic three variable k-map is:



Here are two expressions, $AC + \overline{A}B\overline{C}$ and $AB + \overline{A}C$, represented as k-maps:

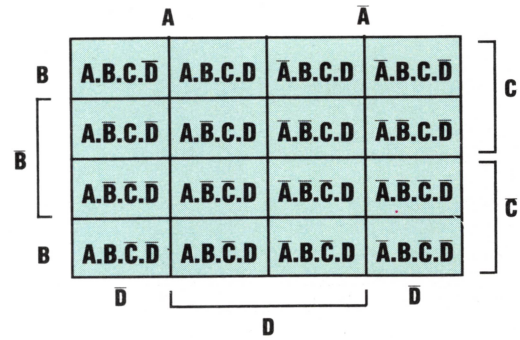


$$ABC+A\overline{B}C+\overline{A}B\overline{C}$$
$$=AC(B+\overline{B})+\overline{A}B\overline{C}$$
$$=AC+\overline{A}B\overline{C}$$

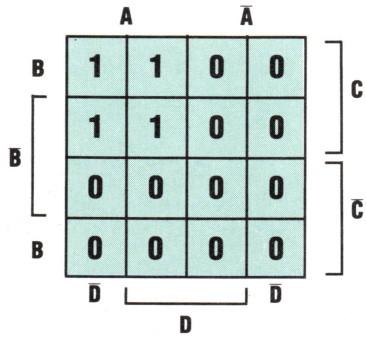
$$ABC+AB\overline{C}+\overline{A}BC+\overline{A}\,\overline{B}C$$
$$=AB(C+\overline{C})+\overline{A}C(B+\overline{B})$$
$$=AB+\overline{A}C$$

Notice that both expressions have been simplified using the Boolean law that a set A ORed with its negation ($\overline{A}$) produces 1 — the Universal or Identity set.

Four Variables: When we start dealing in four variables, the maps get more complex (they have $2^4 = 16$ squares), but nevertheless are quite simple to interpret according to the basic grid:

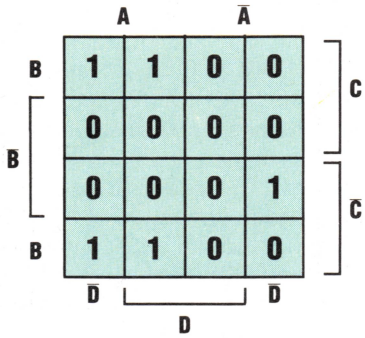


Here is one k-map with an accompanying simplification:



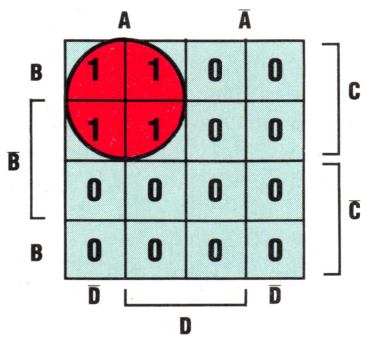
$$
\begin{aligned}
&ABC\overline{D} + ABCD + A\overline{B}C\overline{D} + A\overline{B}CD \\
&= ABC(\overline{D} + D) + A\overline{B}C(\overline{D} + D) \\
&= ABC + A\overline{B}C \\
&= AC(B + \overline{B}) \\
&= AC
\end{aligned}
$$

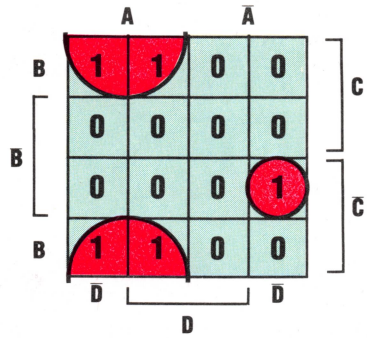
Here is another example:



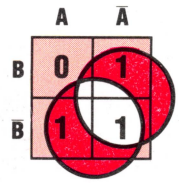
$$
\begin{aligned}
&ABC\overline{D} + ABCD + \overline{A}B\overline{C}\overline{D} + AB\overline{C}\overline{D} \\
&+|AB\overline{C}D \\
&= ABC(\overline{D} + D) + \overline{A}B\overline{C}\overline{D} + AB\overline{C}(\overline{D} + D) \\
&= ABC + \overline{A}B\overline{C}\overline{D} + AB\overline{C} \\
&= AB(C + \overline{C}) + \overline{A}B\overline{C}\overline{D} \\
&= AB + \overline{A}B\overline{C}\overline{D}
\end{aligned}
$$

If we look closely at the arrangement of the ones in these two k-maps we can discover patterns in them. In the first example, all the squares with AC in their expressions have 1 in them. In the second example this is the case with all the AB squares. This suggests that an easier way of simplifying Boolean expressions is simply to inspect a k-map. Consider these:





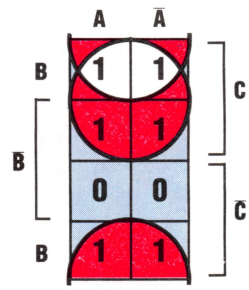
With a little practice it is possible to pick out groups of 2, 4 or 8 ones to form simpler terms. For example, let's consider this expression: $\overline{A}B + A\overline{B} + \overline{A}\overline{B}$.



Using a two variable k-map, we can pick out two groups of ones. One group represents all the NOT(B) cases and the other represents all the NOT(A) cases, so we can simplify the expression to $\overline{A} + \overline{B}$. This expression can be further simplified, using de Morgan's law, to: $\overline{A.B}$. Is it possible to arrive at the conclusion more directly by inspecting the k-map?

A more difficult example involves a three variable expression:

$$ABC + A\overline{B}C + \overline{A}BC + AB\overline{C} + \overline{A}B\overline{C} + \overline{A}\overline{B}C$$



The group of four ones at the top of the k-map represent all the possible cases in which C is true. The top and bottom rows of the map represent all the possible cases in which B is true. Hence the simplified expression is: B + C.

In the next instalment of the course, we will continue our investigation into the use of Karnaugh maps in the simplification of Boolean expressions involving four variables. Then we will show how k-maps are used in the process of circuit design. This will bring together all the aspects of the course discussed so far.

Exercise 4:
Draw up three-variable k-maps to simplify the following Boolean expressions:

a) $\overline{A}.B.C + \overline{A}.\overline{B}.C + \overline{B}.\overline{C} + \overline{A}.B.\overline{C}$

b) $A.\overline{B}.\overline{C} + \overline{A}.B.\overline{C} + A.B.\overline{C}$

COMMANDING CODES

Nobody could call the Commodore's version of the BASIC language advanced, but — like Commodore machines themselves — it has a sturdy simplicity and logic. There may be glaring deficiencies in, for example, its commands but these can usually be compensated for, and its screen editor is still one of the best available.

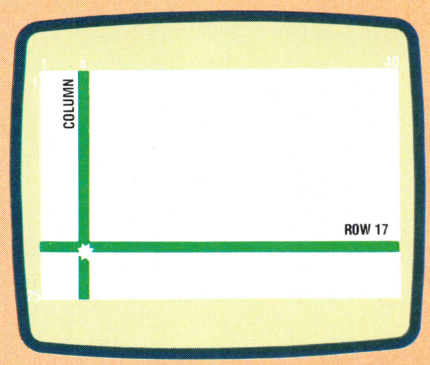
All CBM machines allow long variable names, but only the first two characters of a name are scanned by the interpreter, so variables such as FUJIYAMA and FUTILITY are allowable, but equivalent. Thus the output of this fragment:

```
100 FUJIYAMA=17:FUTILITY=2*FUJIYAMA
200 PRINT FUJIYAMA,FUTILITY
```

is:

```
 34        34
```

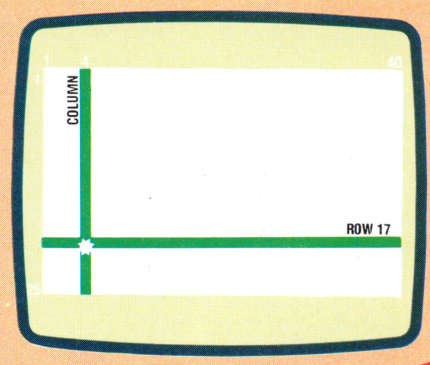
Positioning The Cursor



The ability to include cursor commands in a string quantity can make graphic design on the Commodores easy — especially when coupled with the powerful screen editing command

```
100 PRINT AT(17,4)"*"
```

If Commodore BASIC supported the PRINT AT command, then positioning the cursor would be simple.



Since it doesn't permit this, we must use the programmable cursor feature:

Initialise POSITION$ with a HOME and 24 CRSR DOWNs. Then put the row and column parameters in this expression.

```
50 POSITION$="
100 PRINT LEFT$(POSITION$,17)TAB(4-1)"*"
```

If you have a lot of screen formatting to do, it might be worth putting the cursor positioning command into a subroutine, and then initialising the variables ROW and COLUMN with the screen position required before calling the subroutine

```
50 POSITION$="
100 ROW=17:COLUMN=4:GOSUB 1000:PRINT "*"
500 END
1000 PRINT LEFT$(POSITION$,ROW)TAB(COLUMN-1);:RETURN
```

LIZ DIXON

This applies to all variable types: floating point (e.g. NUMBER), integer (e.g. NUMBER%), string (e.g. NUMBER$), and array (e.g. NUMBER$(62,47)). The variable types are themselves conventional, but on the Vic-20 integer variables are unusable because the machine does not support integer arithmetic; the integer type was retained simply for compatibility with other CBM machines that support integer arithmetic.

An annoying consequence of the variable name rules is that a valid-looking name may be illegal because its first two letters make a reserved word — START, for example, is equivalent to ST as a variable name, and ST is a reserved word.

Array variables may have up to 225 dimensions and are limited in extent only by the amount of RAM available. The first element of any array is element(0), so DIM EX(6) creates an array of seven elements: EX(0), EX(1), EX(2). . .EX(6). The DIM statement is actually unnecessary here, for if the interpreter encounters a single-dimension array variable for which no DIM statement has been executed, a default dimension of 10 is assumed; if the subscript of such an array is greater than 10, then a BAD SUBSCRIPT error will result. This is a convenient facility but it does not encourage good programming practice: the interpreter has to rearrange memory whenever it encounters a DIM statement (or the first reference to an unDIMmed array), so all arrays should be DIMmed at the same time at the start of a program, before any simple variables are employed. No great calamity will occur if this isn't done, but speed of execution will suffer slightly.

Because of the way most BASIC interpreters work, program execution can be speeded up by initialising the most commonly used program variables in their order of importance; this can be done with assignment statements or with the DIM statement. A line such as:

```
10 DIM A$(10,24),K,L,SCORE
```

will have no obvious effect but is a quick way of placing the variables K,L, and SCORE high in the symbol table, thus making them more readily accessible to the interpreter, therefore increasing program execution speed.

A look through the list of BASIC keywords in a CBM user manual (of which more later) reveals a few omissions from, and additions to, the full Microsoft set. The most important omission is probably INKEY$ and the most significant additions are TIME$ and STATUS.

INKEY$, the keyboard-scanning function, is replaced by the statement GET. Like INKEY$, it causes the first character of the keyboard buffer to

be scanned and returns its ASCII value. GET is most often used in statements such as this:

```
150 GET GT$:IF GT$="" THEN 150
```

which should have the effect of halting program execution until a key is pressed, in which case GT$ will contain the character corresponding to the keypress. This may not always be the case because GET scans the keyboard buffer rather than the keyboard itself, so if the buffer contains some characters when GET is performed then program execution will not wait for the user's keypress. That can be demonstrated with this program:

```
50 FOR K=1 to 100:PRINT K:NEXT K
60 PRINT "PRESS ANY KEY"
150 GET GT$:IF GT$="" THEN 150
200 PRINT "YOU PRESSED KEY ";GT$
```

If you run this program you will see the numbers from 1 to 100 appear on the screen, followed by the input message, and then nothing until you press a key. If, however, you press a key while the numbers are being printed, then that keypress will go into the buffer and be found there by the GET statement, so that there will be no pause in program execution. This can be annoying — possibly disastrous in games, for example, where much frantic key-pressing occurs. The answer is to reset the buffer just before the GET statement is executed, by inserting in the program:

```
149 POKE KBPTR,0
```

where KBPTR is the address of the keyboard queue counter (198 in the Commodore 64).

GET does not normally generate a blinking cursor on the screen, but POKE FLASH,0 — where FLASH is the address of the cursor blink enable flag (204 on the Commodore 64) — will supply one.

TIME$ (usually abbreviated to TI$) is the system clock; at power-on it is initialised to '000000' and thereafter indicates the time in hours, minutes and seconds up to '235959' — 23 hours 59 minutes and 59 seconds since initialisation — when it resets to '000000'. It is available to the user like any other variable, and can be set to any legal time such as: TI$="000000" or TI$="084503".

Associated with TI$ is TI, its numeric counterpart. TI contains the time since initialisation in 60ths of a second (called 'jiffys'), so it ranges in value from 0 to 5183999 (60*60*60*24-1). TI is dependent upon TI$ and cannot itself be initialised — you initialise only TI$.

STATUS (abbreviated to ST) is a system-defined variable. When an error is detected at an input/output device the value of ST will be a number indicating the type of error encountered. Typically this would be written as:

```
330 IF ST > 0 THEN GOSUB 30000: REM I/O FILE
    HANDLING ERROR
:
30000 REM ERROR MESSAGES
30100 IF ST=16 PRINT "UNRECOVERABLE READ
      ERROR"
```

CMD is a useful member of the Commodore instruction set. Its effect is to divert output from the screen to a selected output channel. This has many useful applications such as:

```
OPEN 4,4:CMD 4:LIST
```

This LISTs the current program to the printer rather than on the screen; when the LIST is complete you should execute:

```
PRINT#4:CLOSE 4
```

CMD can be used in a program to copy the screen to the printer. Suppose GOSUB 3000 in your program causes a message or some data to be PRINTed on the screen (rather than POKEd into screen memory). In order to copy that display to the printer, type in:

```
OPEN4,4:CMD 4:GOSUB3000:PRINT#4:CLOSE4
```

Although a question mark (?) is acceptable as an abbreviation of the keyword PRINT, you cannot abbreviate the keyword PRINT# to ?# — you must use pR (p followed by shifted r) to do this.

The Commodore keyword abbreviations are as old as the Commodore machines, but to listen to Spectrum owners you would think that Sinclair invented the idea. Almost all keywords can be abbreviated to their initial letter followed by the shifted second letter. When two or more keywords have the same initial two letters then the abbreviation will be the first two lettters and shifted third: READ, RESTORE and RETURN, for example, are abbreviated to rE, reS and reT.

The screen editor, and the robust, user-friendly operating system that underlies it, is the most significant single feature of the Commodore machines. It has been in use since the PET was released and is still the best micro screen editor available. To edit a program line, for example, LIST the line, move the cursor directly to any point in the line, edit the text, and press return. It doesn't matter where the text is on the screen or where the cursor is in the line — when you press return the text on the screen line containing the cursor enters the system as if you'd just typed it.

One subtlety of this editing system is in the copying facility it permits. Suppose you have to enter these two lines:

```
100 IF INT(NUMBER/INDEX—RATE)=5 THEN 3000
200 IF INT (NUMBER/INDEX—RATE)=7 THEN 3800
```

You need type in only the first line and press return; then, with that text on the screen, move the cursor up, change the line number from 100 to 200, change 5 to 7, change 3000 to 3800, and press return again. Line 100 remains untouched in memory, and its edited text on the screen becomes line 200. This process may be repeated as often and as much as you wish. That's only one of the fascinating tricks you can play with the editor, but it demonstrates its ease and directness of use. After the Commodore editor, using other systems feels like playing the piano with mittens on.

SUPER SWINDLER

Jerry Schneider swindled $1,000,000 worth of equipment from the Los Angeles Pacific Telephone and Telegraph company in 1971. Salvaging manuals and equipment from PT & T's waste bins, he posed as a reporter and acquired access codes for PT & T's IBM 360 computer. Discreet orders were made and goods re-sold by Schneider. He served 40 days in prison and then took up as a computer security consultant.

ASSEMBLY LINES

In this part of the Machine Code course, we summarise the main conventions used when dealing with memory, particularly: lo-hi addressing, tokenisation, and the importance of context. We also introduce some differences between Assembly language programs for the 6502 and Z80 microprocessors.

When you RUN a program, the first thing that the Operating System does is to inspect the Start of BASIC Text pointers in order to determine where in memory the program to be executed resides. But to do this, the Operating System has to store the pointers' addresses, so why doesn't the OS simply store the addresses to which they point?

The main reason is flexibility. The Operating System, you will remember, is a permanent program resident in the ROM, and any data that it contains (such as memory addresses) is similarly permanent. Suppose that different versions of the computer are released over a period of time, and that, although it was convenient to have the Start of BASIC Text at byte2048 in Version 1, it becomes necessary to relocate it at byte4096 in Version 2. This will mean that the later machine will not be able to use the Operating System of the earlier version because of the different locations of the BASIC Text Area. Furthermore, new ROMs would have to be developed for each new version of the machine, which is expensive; and software written for one version may not be able to be run on the other. If, however, the Operating System ROMs contain only the pointer addresses, then the same ones can be used for all versions of the machine and only the pointer contents need be changed from model to model. The location of the pointers themselves can remain constant because the Operating System requires a relatively small block of memory for workspace and data storage (typically about 1,000 bytes). Fixing the position of this block — usually the first four pages of memory — and designing or re-designing the system around it does not greatly constrain the design team. On the other hand, having the location of, say, the BASIC Text Area fixed (a block of 3,000 to 40,000 bytes) is a severe restriction.

STANDARD PRACTICE

It is conventional to store addresses in pointers in what is called *lo-hi* form. If byte43 and byte44, for example, are to point to the address 7671 (page 29, offset 247), then byte43 will contain 247 (the offset or lo byte of the address), while byte44 will contain 29 (the page or hi byte of the address). This may seem confusing but it is convenient for the microprocessor. It is also logical in that the lo byte of the address is stored in the lo byte of the pointer, and correspondingly the address hi byte in the pointer hi byte.

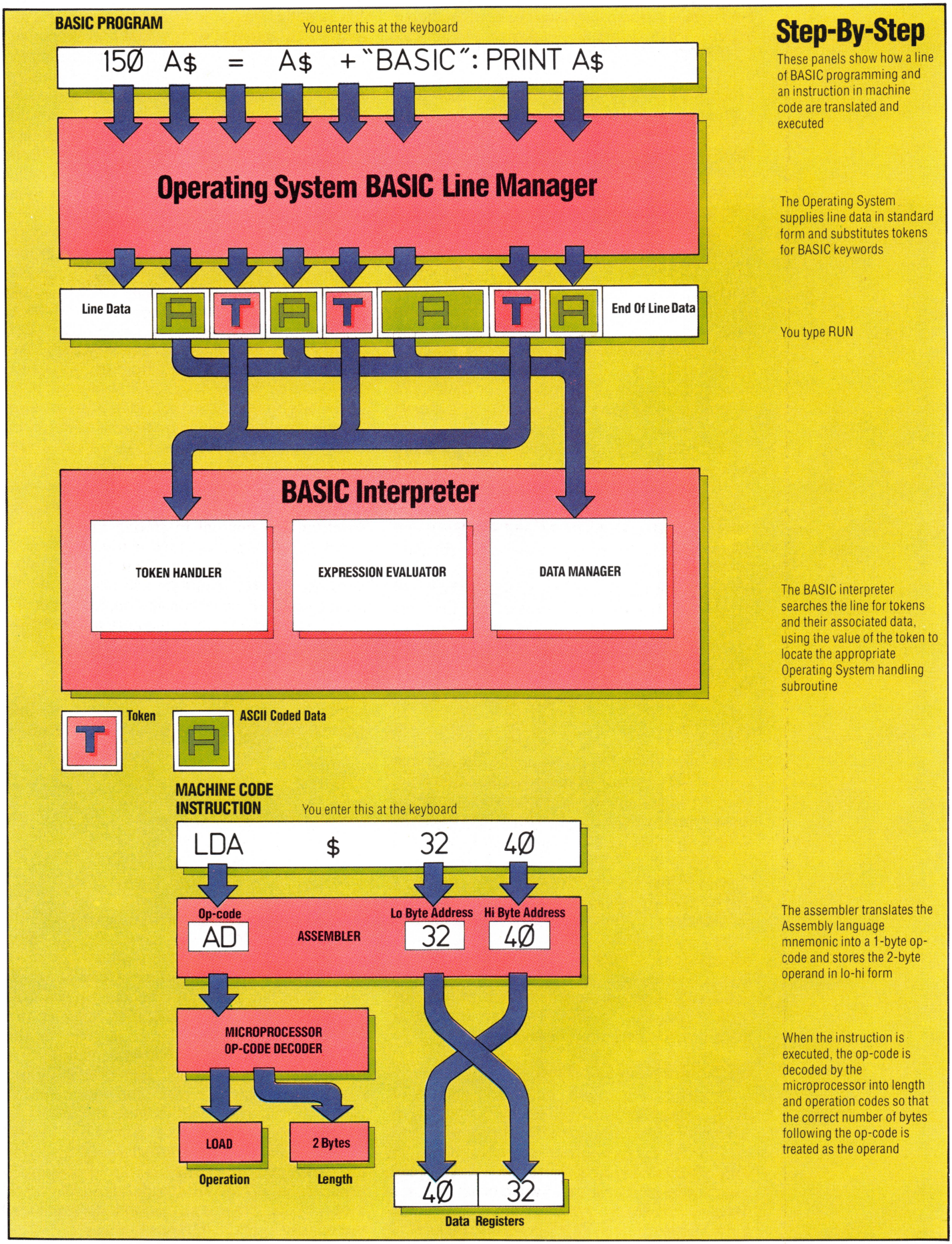
If we repeat the above example using hex rather than decimal numbers, the great advantage of the hexadecimal system can be seen (from now on addresses and other numbers will always be written in hex prefixed by '$'). The pointer bytes are $2B and $2C, and the address to which they point is $1DF7. Therefore, $2B contains F7 (the address lo byte), while $2C contains $1D (the address hi byte). Notice that when the address is in hex the rightmost two hex digits are the lo byte, and the leftmost two digits are the hi byte, which makes much better sense than using decimal numbers.

It's worth remarking that the BBC and Spectrum are unusual in storing program line numbers as two-byte numbers in hi-lo rather than lo-hi form. It's true that these are program parameters rather than byte addresses, but they work against the usual convention, nonetheless.

Another convention of memory addressing is that pointers, although they are two-byte quantities, are often referred to by the address of the lo byte alone. We might say, for example, that byte43 in the Commodore 64 points to the Start of BASIC Text. It is understood here, however, that byte43 and byte44 together are the pointers.

Other things to consider include tokens (see page 76). The significance of these for machine code programmers is two-fold: they represent multi-character English commands (such as PRINT or RESTORE) by single-byte numerical codes; and they use offsets as well. A BASIC command is one word, but executing it is not a single operation for the Operating System. The command PRINT, for example, requires that the data to be printed be found in memory or evaluated, and then sent character-by-character to the screen in ASCII code. These various tasks are carried out by a subroutine of the BASIC interpreter program. When the interpreter encounters the PRINT token in a program line it uses the value of that token to locate and then execute the corresponding subroutine.

Suppose there are only three commands in our version of BASIC: INPUT, PRINT, and STOP; and these are assigned the tokens: $80, $81, and $82 respectively. Furthermore, let's say that the interpreter subroutines that execute these commands start at bytes $D010, $EA97, and $EC00 respectively, and that these three addresses are stored in lo-hi form in the six bytes from $FA00 to



Step-By-Step

These panels show how a line of BASIC programming and an instruction in machine code are translated and executed

The Operating System supplies line data in standard form and substitutes tokens for BASIC keywords

You type RUN

The BASIC interpreter searches the line for tokens and their associated data, using the value of the token to locate the appropriate Operating System handling subroutine

The assembler translates the Assembly language mnemonic into a 1-byte op-code and stores the 2-byte operand in lo-hi form

When the instruction is executed, the op-code is decoded by the microprocessor into length and operation codes so that the correct number of bytes following the op-code is treated as the operand

KEVIN JONES

$FA05, giving us a table of three two-byte pointers. Now when our imaginary interpreter encounters a token — $81, for example — it proceeds to subtract $80 from it, multiplies the result by two, and adds that to $FA00. The final result in this case is $FA02, which is the lo byte of the pointer to the PRINT subroutine. If a token other than $81 had been encountered, then the algorithm described would have returned the pointer address for the corresponding subroutine. In this way the BASIC command PRINT is replaced by a token, $81, which is an offset to a table of pointers that direct the interpreter to the relevant part of its own program.

That's a measure of the 'distance' between BASIC, a so-called high-level language, and machine-code, the low-level language. BASIC looks comprehensible to us because it uses English language code words, algebraic logic and numbers and strings. When we replace the words by tokens, and the rest by ASCII codes, it begins to look a lot more like something a microprocessor can handle — and, as we've seen with tokens, that's almost exactly the case.

The final thing to consider about memory manipulation is the idea of *context*. We've seen in the BASIC Text Area the widespread use of codes — ASCII codes to represent characters and numbers, tokens to represent commands, and (in the Spectrum) special binary codes to represent numeric data (see page 78). All of these codes reduce to binary numbers in the range 00000000 to 11111111 ($00 to $FF, 0 to 255 decimal) contained in single bytes of memory, and interpreted according to their context. In the BASIC Text Area of the Commodore 64, for example, the BASIC program line:

```
200 rem*******left$*********
```

might have three bytes containing the decimal number 200 — once in the link address lo byte, once in the line number lo byte, and once in the token representation of 'left$'. Each byte looks the same as the others, yet means something different. It is only your expectations that tell you how to interpret that value in different places.

This is really where we came in, at the start of the Machine Code course. Then we said that everything stored in a computer is in some sort of machine code. Some of this was familiar (like ASCII codes), some unfamiliar (such as tokens), and some as yet unexplained (such as machine code programs). So let's now start looking at machine code programs themselves.

OPERATION CODES

Programs in machine code are sequences of bytes located anywhere in memory that are a mixture of instructions to the microprocessor, and data for the microprocessor to operate upon. As with all other bytes of memory, it is only the context that can separate the data bytes from the instruction bytes, so we must first consider the format of machine code program instructions.

A machine code instruction begins with a code that identifies the operation to be performed. This is called the *op-code*, or *opc*, and may be one or two bytes in length. The op-code may be a self-sufficient instruction requiring no data, but more usually it is followed by one or two bytes of data. A single byte of data is likely to be a numerical constant or an ASCII code, while two bytes of data following an op-code are always an address (always stored in lo byte/hi byte form). With the above definition we immediately come upon differences between microprocessors: the BBC Micro uses a MOS Tech 6502A, the Commodore 64 uses a MOS Tech 6510 (very similar to the 6502A, so in future we'll talk generally about the 6502 only), and the Spectrum has the Zilog Z80A. MOS Tech and Zilog developed their microprocessors at about the same time — the early 1970s — following the release by Intel of the first microprocessor in 1971. Both the 6502 and Z80 therefore share a design philosophy, but they differ sharply in detail. In particular, Z80 machine codes are completely different from 6502 machine codes. Thus, for example, 6502 op-codes are always one byte long, and may be followed by one or two data bytes or by none; but Z80 op-codes can be two bytes long, followed by one or two data bytes or by none.

When sent to the microprocessor, an op-code is decoded by the CPU's internal program into operation and length codes, and it is this latter information that enables the microprocessor to interpret the bytes following the opc. For example, to the 6502 the sequence of hex bytes:

```
A9 0E 8D 01 4E 60 44 52 41 54
```

represents three instructions, followed by four bytes of ASCII codes. This could be re-written as:

```
A9 0E
8D 01 4E
60
44
52
41
54
```

showing that the first instruction is opc A9, which is always followed by one data byte; the next instruction is opc 8D, which is always followed by two data bytes; while the next is opc 60 which requires no data and causes program execution to branch, so that the following data bytes are not examined by the processor at all. If the microprocessor is sent the first byte, A9, when it is expecting to receive an opc, then everything will function smoothly thereafter. The information in each opc will ensure that the correct number of data bytes for each opc is picked up by the processor, and the following byte will be treated as the next opc. If, however, the processor is expecting an opc and is sent the second byte, 0E, then it will treat this as an opc, with the result that the sequence will be interpreted thus:

```
0E 8D 01
4E 60 44
52
```

meaning: opc 0E, which takes two data bytes; then opc 4E, which also takes two data bytes; then opc 52, which isn't a legal opc, causing the equivalent of a syntax error in the processor. This demonstrates how an initial misunderstanding generates a series of gross logical errors in the program execution.

This also clearly shows some other important points about machine code: it really *is* unfriendly to the user (at least, in the beginning) in that it is difficult to read and write; it is highly sequential with nothing but the order differentiating one instruction from the next; and it is literal as only a machine can be, obeying wrong instructions as readily as it will correct ones, and rejecting only syntax errors.

Some of the unfriendliness can be avoided by writing alphabetic mnemonics instead of the numeric op-codes while the program is being developed, and only resorting to op-codes when the program is actually loaded into memory. These mnemonics constitute the processor Assembly language, and translating them into numerical op-codes is called assembly or assembling. Notice that there is a direct one-to-one correspondence between the set of Assembly language mnemonics, and the set of op-codes: although Assembly language is a higher-level language than machine code, the difference is minimal.

If we rewrite the machine code fragment above in 6502 Assembly language, then it looks like this:

```
0000 A9 0E      LDA #$0E
0002 8D 01 4E   STA $4E01
0005 60         RTS
```

while the same sequence of operations in Z80 Assembly language looks like this:

```
0000 3E 0E      LD A,$0E
0002 32 01 4E   LD ($4E01),A
0005 C9         RET
```

The first column shows the hex address in memory of the first byte of the line — the opc A9 in the 6502 list, for example, is in byte0; the page byte 4E in both lists is in byte4, and so on. The next column may contain one, two or three bytes, and shows the machine code listing. The third column starts with an Assembly language mnemonic, and shows the Assembly language version of the machine code. Don't bother trying to puzzle it all out now, it's enough that you've seen an Assembly language list, and can observe the differences and similarities between the Z80 and 6502 versions. You might also notice that the address in the second line appears in conventional lo-hi form in machine code, but 'normal' hi-lo form in Assembly language.

In the next instalment of the course we'll start to examine op-codes in detail, and take a look at the architecture of the microprocessor.

Hexadecimal Convertor

To convert the Mempeek program of page 59 so that byte contents are displayed in hexadecimal rather than decimal, make the following changes:

BBC Micro

Add:

```
3000 DEF PROCHXPRINT(DECNUM)
3100 LOCAL X$
3200 X$="0123456789ABCDEF"
3300 HB=INT(DECNUM/16):LB=DECNUM-HB*16
3400 B$=MID$(X$,HB+1,1)+MID$(X$,LB+1,1)+"   "
3500 PRINT B$;
3600 ENDPROC
```

and change line 600 to:

```
600 PROCHXPRINT(PK%)
```

Spectrum

Add:

```
10 LET X$="0123456789ABCDEF"
3000 REM******HEX BYTE S/R********
3100 LET HB=INT (PK/16): LET LB=PK-HB*16
3200 LET B$=X$(HB+1)+X$(LB+1)+"   "
3300 PRINT B$;
3400 RETURN
```

and change line 600 to:

```
600 GOSUB 3000
```

Commodore 64

Add:

```
10 LET X$="0123456789ABCDEF"
3000 REM******HEX BYTE S/R********
3100 HB=INT(PK/16):LB=PK-HB*16
3200 B$=MID$(X$,HB+1,1)+MID$(X$,LB+1,1)+"   "
3300 PRINT B$;
3400 RETURN
```

and change line 600 to:

```
600 GOSUB 3000
```

These changes will cause the contents of the memory to be displayed in hexadecimal. The start address and number of bytes should still be entered in decimal

FUTURE INVESTMENT

Xerox, the world's biggest seller of reprographic equipment, were keen to enter the area of office automation. Their reputation was already firmly established — so much so that people refer to the activity of photocopying as 'xeroxing' — and the new impetus aimed to extend Xerox's pre-eminence to other office machines.

In the early 1970s, the Xerox corporation planned a full-scale research programme to realise a dream — to have information available on tap in the office, like electricity or running water. Xerox created a new research team with an open brief, and encouraged maximum freedom of operation by setting it up in Palo Alto, California, on the far side of the USA from Xerox's worldwide headquarters in Rochester, New Hampshire.

The move to Santa Clara county, California, was a fruitful one. Located close to the campus of Stanford University, which had a thriving computer science department specialising in artificial intelligence research, the Palo Alto Research Center (PARC) attracted some of the best brains in computing. In this close community, talented students were able to move smoothly from academic to commercial research. PARC became the centre of the computer culture, spawning a jargon understood only by initiates. Several Xerox products were given jokey names while under development. The 820 series of micros, for example, was code-named 'Worm' on the grounds that it was going to 'eat Apple'.

The main impetus of the new research team was directed at developing a local area network (LAN). This term is now commonplace, but when Xerox made its first experimental network, in Hawaii in the late 1960s, it was a revolutionary concept. Connections between a mainframe machine and a terminal required expensive cabling for high-speed communications, and there were problems with cable runs longer than 20 metres (70 feet). The public switched telephone network could be used but this restricted exchange of data to 9,600 baud.

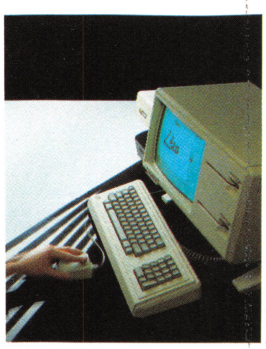
The aim at Palo Alto was for a network with reasonable speed that would link smaller computers together so that the user would have local computing power for his own machine, as well as access to larger computers, big disk stores, and other expensive peripherals such as plotters and printers. This was the basis of the Ethernet LAN concept.

In the Ethernet system, connections were made with ordinary co-axial cable, which is capable of carrying 10 million bits per second and is suitable for carrying digitised sound and graphics information as well as data. Moreover, the system could run up to 500 metres (1700 feet) without the need for repeater amplifiers. Any new device could be plugged in by simply tapping into the existing net, allowing maximum flexibility.

The physical net is passive: data of whatever kind is broadcast around the network and a *transceiver* acts as the front end of each device, determining whether the message is intended for that device. If it is, the transceiver decodes the message and presents it in a form that the device — whether microcomputer, printer, plotter or whatever — can use.

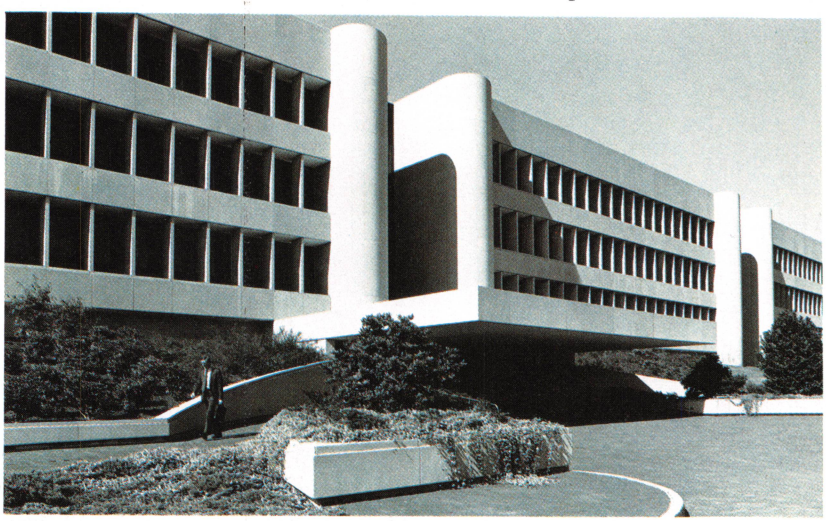
By the mid-1970s Ethernet was working. Xerox felt that if it could enlist the help of other manufacturers, the system would become a standard for communication among computers. It took its designs to IBM, who declined to participate. The Digital Equipment Corporation, however, were keen to become involved. In 1975, Xerox also secured the all-important co-operation of chip manufacturer Intel, which started to build the transceiver chip.

Ethernet was put on trial in an experimental office and factory complex in Sweden and after successful testing was adopted by other manufacturers. It has now become an official international standard and manufacturers such as Hewlett-Packard, ICL in the UK, Siemens in Germany and Olivetti in Italy have all decided to adopt it. Xerox fostered acceptance of the standard by selling the blueprints for a single fee of $1,000. All Xerox products, from microcomputers and typewriters to laser printers — another PARC invention — can be connected to Ethernet.



CHRIS STEVENS

Early Lisa
One of the major triumphs of the PARC was the development of STAR, a programming system that uses the SMALLTALK language. STAR operates by combining programs and data in the same file for processing. Apple's Lisa technology owes much to this approach — in fact, most of Lisa's development team were recruited from the PARC



Mentathlete

Home computers. Do they send your brain to sleep – or keep your mind on its toes?

At Sinclair, we're in no doubt. To us, a home computer is a mental gym, as important an aid to mental fitness as a set of weights to a body-builder.

Provided, of course, it offers a whole battery of genuine mental challenges.

The Spectrum does just that.

Its education programs turn boring chores into absorbing contests – not learning to spell 'acquiescent', but rescuing a princess from a sorcerer in colour, sound, and movement!

The arcade games would test an all-night arcade freak – they're very fast, very complex, very stimulating.

And the mind-stretchers are truly fiendish. Adventure games that very few people in the world have cracked. Chess to grand master standards. Flight simulation with a cockpit full of instruments operating independently. Genuine 3D computer design.

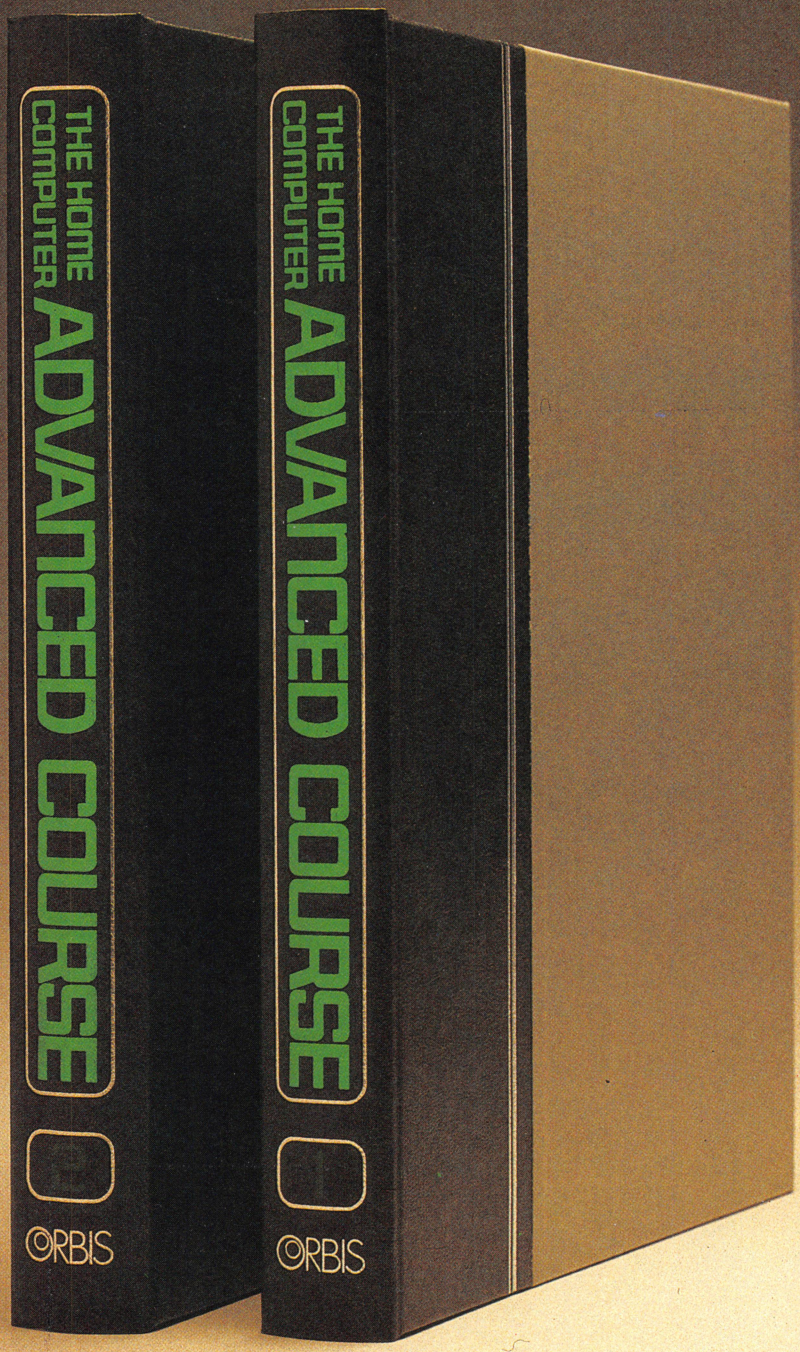
No other home computer in the world can match the Spectrum challenge – because no other computer has so much software of such outstanding quality to run.

For the Mentathletes of today and tomorrow, the Sinclair Spectrum is gym, apparatus and training schedule, in one neat package. And you can buy one for under £100.

sinclair

THE HOME COMPUTER ADVANCED COURSE

WE HAVE DESIGNED BINDERS SPECIALLY TO KEEP YOUR COPIES OF THE 'ADVANCED COURSE' IN GOOD ORDER.



All you have to do is complete the reply-paid order form opposite – tick the box and post the card today – **no stamp necessary!**

By choosing a standing order, you will be sent the first volume free along with the second binder for £3.95. The invoice for this amount will be with the binder. We will then send you your binders every twelve weeks – as you need them.

Important: This offer is open only whilst stocks last and only one free binder may be sent to each purchaser who places a Standing Order. Please allow 28 days for delivery.

Overseas readers: This free binder offer applies to readers in the UK, Eire and Australia only. Readers in Australia should complete the special loose insert in issue 1 and see additional binder information on the inside front cover. Readers in New Zealand and South Africa and some other countries can obtain binders now. For details please see inside the front cover. Binders may be subject to import duty and/or local tax.

The Orbis Guarantee: If you are not entirely satisfied you may return the binder(s) to us within 14 days and cancel your Standing Order. You are then under no obligation to pay and no further binders will be sent except upon request.

PLACE A STANDING ORDER TODAY.